Les ingrédients des algorithmes

> PAR GILLES DOWEK, THIERRY VIÉVILLE ET EMMANUEL BACCELLI, CHERCHEURS À L'INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (INRIA), JEAN-PIERRE ARCHAMBAULT ET BENJAMIN WACK, ENSEIGNANTS

A Introduction

Un ordinateur peut effectuer des opérations très variées, sur des types de données tout aussi variés : des nombres, des lettres, des images, des sons, des textes, des vidéos, comme le montre la séquence pédagogique précédente. Il peut être utilisé pour retoucher une photographie, la mettre sur un blog ou un site Web, la conserver dans un album, etc. Un ordinateur est une machine polyvalente : tout ce qui s'automatise peut être programmé sur un ordinateur. À l'inverse des machines à café ou des aspirateurs, qui ne servent qu'à une seule chose : faire le café ou aspirer la poussière.

En revanche, un ordinateur ne « sait » quasiment rien faire. Il doit être « programmé » pour retoucher une photographie, la mettre sur un blog ou un site Web. Cette notion de programme est également ce qui distingue l'ordinateur de certaines machines, comme les calculatrices non programmables, qui donnent une illusion de polyvalence dans la mesure où elles peuvent effectuer des tâches multiples. Cependant, ces tâches sont fixées une fois pour toutes : on ne peut pas programmer une calculatrice pour lire une vidéo, alors que l'on peut programmer un ordinateur pour faire tout ce que fait une calculatrice.

En quoi consiste l'écriture d'un tel programme ? Le plus important est de bien spécifier ce que l'on cherche à faire. Ce qui compte, ici, c'est la méthode mise en œuvre : l'algorithme. Il faut ensuite l'écrire de façon à le transmettre à l'ordinateur. C'est pour cela qu'on utilise un langage de programmation. Les documents mis en ligne sur le site *Interstices* (www.interstices.info), « Algorithme, mode d'emploi » et « Demandez le programme », offrent une introduction à ce sujet. Voir également http://javascool.gforge.inria.fr/proglet : le site propose une activité scolaire qui met en application les éléments décrits dans cette séquence.

B Algorithme... une recette de cuisine ?

On appelle « algorithme » la méthode, la façon systématique de procéder, pour faire quelque chose : trier des objets, situer des villes sur une carte, multiplier deux nombres, extraire une racine carrée, chercher un mot dans le dictionnaire, etc. Il se trouve que certaines actions mécaniques – peut-être toutes – se prêtent bien à cette décortication. On peut les décrire de manière générale, identifier des procédures, des suites d'actions ou de manipulations précises à accomplir séquentiellement. C'est cela, un algorithme : le concept qui traduit la notion intuitive de procédé systématique, applicable mécaniquement, sans réfléchir, en suivant un mode d'emploi précis.

Un algorithme, c'est donc « presque » une recette de cuisine. Prenons l'exemple d'un quatre-quarts au citron ou au chocolat. Il suffit d'utiliser deux œufs, le même poids de farine, de beurre et de sucre, d'ajouter le parfum, de mélanger le tout et de mettre au four une « petite » demi-heure. Mais que se passerait-il si nous confiions cette recette à un dispositif mécanique dépourvu d'intelligence (robot, ordinateur, automate) ? Il réaliserait notre recette *exactement* comme nous venons de le spécifier. « Une petite demi-heure dans le four » ? Voilà notre dispositif mécanique bloqué : petite ? Par rapport à quoi ? De combien ? Sans compter que personne n'a précisé d'allumer le four, ni de verser le mélange dans un moule !

En résumé, toute instruction ambiguë, incomplètement spécifiée, va échouer. Un algorithme est donc bien une recette de cuisine... mais c'est une recette à définir, à spécifier. En clair, un dispositif mécanique ne possède aucune information contextuelle, aucune connaissance préalable, aucun de ces éléments qui font la richesse de l'intelligence humaine. Il faut donc que chaque étape soit entièrement et explicitement spécifiée, dans ses moindres détails.

C De retour en cuisine

Nous allons demander au robot d'exécuter la séquence de toutes les opérations (par exemple : ouvrir le vaisselier, sortir le grand compotier, ouvrir le réfrigérateur, se tourner vers le compartiment supérieur de la porte, sortir le beurre, ouvrir le placard, se tourner vers la troisième étagère à droite, prendre la farine, etc.). Notre algorithme est donc un chemin à parcourir pas à pas, une **séquence d'instructions.** Pour peu que chaque instruction ait été bien programmée précédemment, nous commençons à voir apparaître la pâte du gâteau. À condition que tout se passe exactement comme prévu !

Que faire si le compotier n'est pas à sa place, ou s'il ne reste plus de beurre ? « S'il n'y a plus de beurre, il suffit de prendre de la margarine, sinon il vaut mieux arrêter de faire le gâteau… » Voilà que notre recette n'est plus une simple séquence d'instructions, ce n'est plus un simple chemin, mais un itinéraire avec des « carrefours », où le choix du chemin se fait en fonction d'une condition (pas de beurre, ou ni beurre ni margarine). L'algorithme est donc un réseau d'**instructions conditionnelles,** à parcourir au fur et à mesure, en bifurquant à chaque condition.

Le quatre-quarts est en bonne voie. Mais comment expliquer au robot qu'il faut tourner une trentaine de fois les ingrédients dans le compotier sans devoir recopier une bonne trentaine de fois l'instruction ? Car le but n'est pas de tourner trente fois, mais plutôt de tourner jusqu'à ce que la pâte soit bien homogène. Nous avons alors besoin d'une autre construction, une boucle de la forme : « Tant que la pâte n'est pas homogène, tourner les ingrédients dans le compotier. » Avec cette **boucle d'instruction,** il est possible de faire durer ou répéter une opération autant de fois que nécessaire. De même, nous pouvons écrire : « Cuire au four tant que la pâte reste liquide » ou « Fabriquer 200 quatre-quarts pour tout le quartier », en l'exprimant de manière concise.

Enfin, nous voilà avec un quatre-quarts au citron ou au chocolat. Va-t-il falloir réécrire la recette complète pour *chaque* parfum ? Alors que seule l'étape « ajouter trois zestes de citron » ou « ajouter 50 grammes de poudre de chocolat » change d'une recette à l'autre ? Certes pas ; pour l'éviter, nous avons besoin d'introduire la notion de **variable** ou de *paramètre*. La recette doit être *paramétrée* par la variable « parfum » et en fonction de sa *valeur* (« citron », « chocolat »). Une instruction conditionnelle changera juste la ligne de la recette liée au parfum. Nous obtenons ainsi, exprimée de manière concise, la recette de tous les quatre-quarts possibles, qui ne diffèrent que par leur parfum.

À présent, prenons un peu de recul. Nous avons utilisé des instructions comme « ouvrir le réfrigérateur » ou « tourner les ingrédients dans le compotier », qui ont sûrement dû être programmées par une autre personne, et nous avons produit l'instruction « faire un quatre-quarts au goût de $parfum » (le signe $ indique qu'il s'agit d'une variable), où « $parfum » prend la valeur « citron » ou « chocolat ». Bref, nous avons utilisé des fonctionnalités prédéfinies, des « briques de base », pour créer une autre fonctionnalité pouvant être utilisée par une personne qui prépare un menu, par exemple. Cette dernière construction, qui consiste à regrouper un bloc d'instructions dans une **fonction,** va permettre de réutiliser chaque fonctionnalité, comme dans un Lego, pour réaliser une construction logicielle. Ce qui est très intéressant, ici, c'est qu'il suffit de ces cinq ingrédients pour décrire tous les algorithmes de façon efficace. N'oublions pas toutefois une autre différence importante entre les recettes de cuisine et les programmes. Ces derniers sont destinés à être lus par des humains, mais aussi par des ordinateurs. Ils doivent donc être écrits dans des langages très particuliers : les langages de programmation, qui conditionnent la façon de présenter les ingrédients.

D Et les *bugs*, dans tout ça ?

Nous parlons de « recette de cuisine », mais que se passerait-il si l'on se trompait ? Le gâteau serait un peu trop mou ou un peu trop sucré, mais, pour un algorithme, ce serait très probablement la catastrophe absolue ! Voici un exemple : il s'agit de calculer la factorielle d'un nombre *n* (ici, comme souvent dans les notations informatiques, les instructions se placent entre des accolades et se terminent par un point-virgule, l'astérisque représentant la multiplication) :

```
factorielle (entier n) {
    si (n = 0) {
        renvoyer 1 ;
    } sinon {
        renvoyer n * factorielle (n - 1) ;
}
```

Cette formule sert bien à calculer la factorielle, puisqu'elle implémente la définition par récurrence de la factorielle, telle qu'elle est présentée dans nos manuels scolaires, à savoir : factorielle (0) = 1 et factorielle (n) = n * factorielle (n - 1). Si n = 0, le résultat renvoyé est donc 1, sinon le résultat renvoyé est calculé selon la deuxième formule. Avec les ingrédients précédents, on peut proposer une autre implémentation équivalente, c'est-à-dire noter *f* la variable intermédiaire qui va prendre les valeurs du calcul, et écrire le code suivant :

```
entier f = 1 ;
 tant que (n n'est pas nul)  {
    f = f * n ;
    n = n - 1 ;
 }
```

Cette méthode permet de calculer : factorielle (n) = n * (n - 1) * etc. * 1, donc exactement la factorielle que nous connaissons, en s'arrêtant lorsque $n = 0$, ce qui semble parfait… tant que n est positif. S'il est négatif, l'un ou l'autre code va se mettre à calculer factorielle (n) pour les valeurs suivantes, en partant de $n = -1$: $n = -1, -2, -3, -4$… sans s'arrêter ! C'est alors au hasard de décider de ce qui va arriver : ordinateur bloqué, calcul qui dure jusqu'à ce que les batteries soient vides, etc. Bienvenue devant ce qui s'appelle un « *bug* ». En effet, la fonction factorielle n'est que partiellement définie : pour $n \geqslant 0$, tout est bien spécifié, mais pour $n < 0$, rien n'est dit. L'élève reçoit un 0 à sa copie s'il commet l'étourderie ; le programme informatique, lui, boucle à l'infini. Dans ce cas précis, il y a plusieurs parades possibles : soit vérifier que $n \geqslant 0$ avant de lancer le calcul soit modifier le code en écrivant factorielle (entier n), si $n \leqslant 0$ alors renvoyer 1, sinon renvoyer n * factorielle (n - 1), qui renvoie la valeur 1 pour tous les entiers n négatifs ou nuls. En un sens, il s'agit d'étendre la fonction factorielle aux nombres négatifs.

Au-delà de cet exemple, comme nous l'explique Gérard Berry dans son cours au Collège de France sur les sciences numériques, et comme l'enseigne Gilles Dowek dans son cours d'informatique pour des professeurs de mathématiques de lycée, il existe des méthodes formelles qui adoptent une méthodologie plus proche des mathématiques (www.inria.fr/rocquencourt/ressources/multimedia/formation-informatique-et-objets-numerique). Elles permettent ainsi d'effectuer de nombreux calculs sur ces programmes, devenus à leur tour l'objet de manipulations informatiques.

Une technique fondamentale, nommée « interprétation abstraite » – développée à l'École normale supérieure par l'équipe de Patrick Cousot –, permet de détecter automatiquement des *bugs*, comme celui d'Ariane, identifié par Gilles Kahn : lorsque le premier vol d'Ariane 5, en 1996, a explosé et détruit pour 500 millions de dollars de satellites, la faute était due à un morceau de programme qui convertissait une valeur. Créé pour Ariane 4, il a été réutilisé tel quel. Or la valeur à convertir, beaucoup plus grande dans le cas d'Ariane 5, a donné un résultat aberrant et des ordres de braquage maximaux. L'interprétation abstraite évite surtout que de tels *bugs* ne se reproduisent ; elle a été appliquée avec succès sur les logiciels de vol de l'Airbus A380. D'autres techniques offrent la possibilité d'aller jusqu'à la vérification mathématique de propriétés dites de sûreté (« L'ascenseur ne pourra jamais voyager la porte ouverte. ») ou de vivacité (« L'ascenseur finira par prendre tous les passagers qui l'attendent. »). De telles vérifications formelles se font en calculant symboliquement le gigantesque ensemble des exécutions possibles, sans les effectuer explicitement.

Les méthodes formelles que nous allons évoquer permettent ainsi de diminuer sensiblement le nombre de *bugs* dès la conception, puis de trouver des *bugs* particulièrement sournois, hors de portée des tests manuels ou aléatoires ; même si elles ne suffisent pas à elles seules à assurer la sécurité requise pour tous les logiciels du commerce, à cause de leur taille et de leur nombre grandissants, puisque le problème est identifié comme étant indécidable mécaniquement. Il s'agit donc d'exécuter des algorithmes, de vérification et de preuve, sur les données que sont les algorithmes à vérifier ou à prouver. De ce fait, il est nécessaire que ces derniers puissent être exprimés sous une forme interprétable sans ambiguïté. Comment allons-nous donc définir formellement nos ingrédients ?

E De l'algorithme au programme… une notation rigoureuse ?

Les différents langages de programmation sont organisés autour d'un petit nombre de fonctionnalités présentes dans de nombreux langages, relativement stables depuis des décennies, et que l'on peut décrire simplement avec les outils adéquats (affectation, séquence, test, boucle, fonction, récursivité, enregistrement, cellule, module, objet, etc.). Le *noyau* de la plupart des langages est justement constitué des cinq ingrédients que nous détaillons pp. 41-42. Pour chacune de ces fonctionnalités, nous allons décrire :

- sa syntaxe : *la manière dont cette instruction s'écrit.* Ne perdons pas de vue que la question importante est de savoir de quoi est constituée cette instruction, quels sont ses paramètres, etc. Le fait de savoir comment cela s'écrit dans le langage particulier que l'on utilise est accessoire. De même, lorsqu'on apprend à conduire une voiture, l'important est de connaître la signification des mots « frein », « levier de vitesse », etc., et de pouvoir s'en servir. Que ces commandes aient tel ou tel aspect sur une Twingo ou une Mercedes est accessoire ;
- sa sémantique : *ce qui se passe quand on l'exécute.* Détaillons ce point.

Une instruction – ou un groupe d'instructions – a pour but de transformer un *état*, c'est-à-dire la *valeur* de chacune de ses variables. L'état indique donc la valeur de chaque variable à un instant donné. Par exemple, si le programme a trois variables x, y, z qui représentent trois « boîtes » dans lesquelles nous pouvons ranger des valeurs, une instruction va modifier les valeurs de ces boîtes lors de son exécution. De modification en modification, un programme fera passer l'état de sa valeur initiale à sa valeur finale.

Un programme a bien sûr des entrées et des sorties : les sorties sont des boîtes dont les valeurs sont visibles d'un dispositif externe ; les entrées sont des variables dont les valeurs ont été prédéfinies par l'utilisateur ou un dispositif externe. Pour formaliser la sémantique d'une instruction, les informaticiens et les logiciens utilisent habituellement des notations mathématiques, car elles ont l'avantage d'être concises et sans ambiguïté. Nous vous les présentons ci-dessous :

- l'état est représenté par une liste d'équations : [$x = 12$, y = « Hi », $z = 0$] qui donnent la valeur des variables ;
- pour une instruction p, la construction Σ (p, s) = s' signifie qu'elle transforme un état s en un état s'.

Nous sommes maintenant en mesure de détailler chaque instruction.

F Affectation d'une valeur à une variable

La première action est de donner des valeurs aux variables : c'est l'affectation.

MANIÈRE D'ÉCRIRE L'INSTRUCTION

L'affectation est constituée d'une variable x et d'une valeur v. Selon les langages, elle s'écrit $x = v$, $x := v$, $x = v$; etc. Par exemple : $x = 15$. Informellement : la variable x prend comme valeur v. Sa valeur précédente est effacée ; elle gardera toujours la valeur v tant qu'elle ne sera pas changée par une autre affectation de la variable x. L'affectation est donc une « action » : celle d'affecter à la variable x la valeur v. Il faut aussi bien distinguer l'affectation de l'égalité (souvent notée par un double signe ==). Écrire $x == 15$ revient à poser la question : « Est-ce que x est égal à 15 ? » $x == 15$ est vrai si x est égal à 15, faux sinon. Écrire $x = 15$ revient à donner l'ordre que « désormais x sera égal à 15 » !

SENS DE L'INSTRUCTION

Il est facile de comprendre ce que fait cette instruction : elle change l'état en lui ajoutant le fait que désormais $x = v$. Avec les notations mathématiques, nous pouvons le noter ainsi : $\Sigma\ (x = v, s) = s + [x = v]$. Cela signifie que l'on ajoute à l'état la nouvelle équation qui définit la valeur de la variable affectée.

G Fonctions et expressions

La valeur que nous donnons à une variable n'est pas forcément une constante, car nous voulons aussi évaluer des expressions : écrire, par exemple, $x = 12 + z / 2$ pour calculer x à partir de z.

MANIÈRE D'ÉCRIRE UNE EXPRESSION

Une expression numérique (c'est-à-dire qui porte sur des nombres) se construit comme sur une calculette, avec des additions (par exemple : 2 + 3), des soustractions, des multiplications (en utilisant le symbole *) et des divisions (en utilisant le symbole /). Il existe d'autres symboles algébriques que nous n'utiliserons pas ici : il suffit de retenir qu'ils s'écrivent comme pour une calculette. Par exemple, si nous écrivons $2 + (x / 3)$ ou $(2 + x) / 3$, ce ne sont pas les mêmes expressions, puisque le calcul se fait différemment selon les parenthèses.

Une expression logique (c'est-à-dire qui renvoie la valeur « vrai » ou « faux ») se construit avec :
- des comparaisons pour tester si deux valeurs sont égales ou non, si une valeur numérique est plus petite qu'une autre, etc. ;
- des opérateurs logiques comme « et » (noté &&), « ou » (notée ||), ainsi que la négation (notée !).

On peut aussi manipuler des chaînes de caractères, notées entre guillemets, et les concaténer (par exemple, « pa » et « py » se concatènent en écrivant « pa » + « py », ce qui donne « papy ») ou les comparer, en utilisant la fonction equal ( ) : l'expression equal (« pa », « py ») renverra la valeur « faux », puisque les chaînes de caractères « pa » et « py » ne sont pas égales, tandis que equal (« pa », « pa ») renverra la valeur « vrai », puisque ce sont les mêmes valeurs.

D'autres fonctions que les opérations numériques ou logiques peuvent être utilisées. Elles s'écrivent en utilisant des parenthèses : $f(x)$ correspondant à une fonction de nom f, qui prend comme argument en entrée x et va renvoyer un résultat. Par exemple : $f(x) = x * x$ calcule $x * x$, c'est-à-dire le carré de x. Les fonctions peuvent avoir plusieurs arguments, comme $g(x, y, z)$, qui en a trois. Il est d'ailleurs possible de définir soi-même de nouvelles fonctions.

SENS D'UNE EXPRESSION

Pour une expression e (par exemple : $12 + 6 / 2$), nous utilisons une nouvelle notation $\Theta\ (e, s) = r$ pour signifier que la valeur du résultat de son évaluation est r (ici, 15). Il faut donc bien distinguer une expression ($12 + 6 / 2$), une valeur (15) et une affectation ($x = 12 + 6 / 2$).

La valeur d'une expression dépend de l'état. Par exemple, l'expression $12 + z / 2$ dépend de la valeur de z. Ce qui donne, avec les notations mathématiques : $\Theta\ (12 + z / 2, [z = 0]) = 12$ ou $\Theta\ (12 + z / 2, [z = 6]) = 15$. Nous pouvons alors définir l'affectation d'une expression à une variable, avec ces notations : $\Sigma\ (x = e, s) = s + [x = \Theta\ (e, s)]$. Cela signifie que la variable prend comme valeur le résultat de l'évaluation de l'expression.

L'instruction « tourner les pages une à une jusqu'à trouver le pays » s'écrit comme suit :

```
trouver (pays) {
    page = début ;
    tant que page <= fin
        si dictionnaire (page) = pays alors renvoyer page
        page = page +1
    renvoyer « pas trouvé »
}
```

c'est-à-dire énumérer les pages du début à la fin du dictionnaire, ce qui est implémenté à travers la boucle :

```
    page = début ;
    tant que page < = fin
    .../...
    page = page +1
```

dont on sort en renvoyant la page du dictionnaire correspondant au pays si elle est trouvée, comme le spécifie ce test de comparaison :

```
    si
        dictionnaire (page) = pays
    alors
        renvoyer page
```

et dont on sort, à la fin, en signalant ne pas avoir trouvé la page.

Voilà un algorithme qui fonctionne parfaitement. Il coûtera une étape de calcul (ici, un test de comparaison) si le pays est au début du dictionnaire… et 195 étapes s'il se trouve à la fin. Dans le cas présent, le dictionnaire contient N items (N = 195), répartis de manière uniforme ; on voit donc que, de 1 à N étapes de calcul, il y en aura environ C = N / 2 à prévoir. Nous aurions pu énumérer les pages dans un autre ordre (de la fin au début) ou toutes les pages paires, puis les impaires, mais cela n'aurait rien changé : pourvu que nous énumérions toutes les pages, nous sommes certains de trouver le pays recherché, et ce sera alors toujours avec un coût de l'ordre de N étapes de calcul.

PEUT-ON ÊTRE PLUS EFFICACE ?

Si, dans le dictionnaire, les pays n'étaient pas classés par ordre alphabétique mais dans le désordre, la réponse serait négative : nous serions, de fait, obligés de tourner toutes les pages pour être certains de trouver le pays recherché. Mais voilà que le dictionnaire est ordonné de A à Z ; il s'agit là d'une propriété que nous pourrions exploiter. En effet, si nous ouvrons le dictionnaire au milieu, sur la page de la France, par exemple, tandis que nous cherchons l'Albanie, nous voyons que ce n'est pas la bonne page mais nous apprenons quelque chose de plus : puisque l'Albanie est avant la France, sa page est donc forcément située dans la moitié gauche du dictionnaire ; nous n'avons donc plus à chercher dans toutes les pages de droite, mais uniquement dans celles de gauche. Nous avons en une opération réduit notre espace de recherche de moitié. D'environ 200 pages, il ne nous restera plus que 100 pages à explorer. Puis, en reprenant le même procédé, 50 pages, 25 pages, 12 ou 13 pages, 6 ou 7 pages, 3 ou 4 pages, 1 ou 2 pages, et le pays sera trouvé.

Si nous calculons, nous voyons qu'il y a eu uniquement C = 7 ou 8 étapes de calcul ! Ce processus, qui consiste, à chaque étape, à couper en deux parties égales l'espace de recherche, est appelé « dichotomie » (« couper en deux », en grec). Intuitivement, on se rend compte que c'est bien en coupant en deux parties égales que nous sommes sûrs d'avoir des deux côtés un espace de recherche minimal à l'étape suivante (en parts inégales, la malchance pourrait nous conduire à rechercher dans un espace plus grand). Il faut comprendre que le gain est immense. Nous donnons dans le tableau ci-dessous le nombre C d'étapes de calcul en fonction de quelques nombres N d'items :

N = 2C	1	2	4	8	16	…	256	…	1024	…	un million	…	un milliard
C = log2(N)	0	1	2	3	4	…	8	…	10	…	20	…	30

Le fait de diviser en deux, de manière itérative, permet notamment de rechercher en un maximum de 26 étapes le nom d'une personne dans l'annuaire des 60 millions de Français. Il existe plusieurs manières de programmer ce mécanisme. En voici une où nous restreignons l'espace de recherche entre deux valeurs, « min » et « max » :

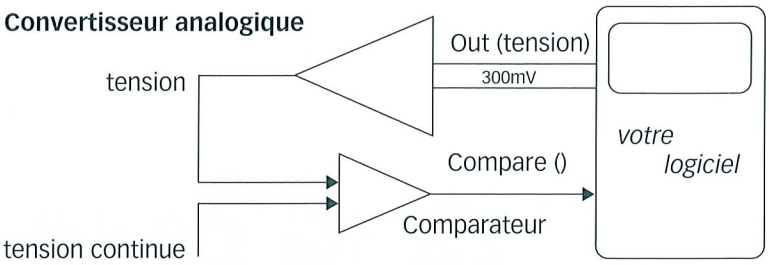
```
trouver (pays) {
    min = début ; max = fin ;
    répéter
        milieu = (min + max) / 2 ;
        si dictionnaire (milieu) = pays alors renvoyer milieu
        si min = max alors renvoyer « pas trouvé »
        si dictionnaire (milieu) < pays alors min = milieu
        si dictionnaire (milieu) > pays alors max = milieu
}
```

où l'on remarque que la quantité « max-min » vaut « fin-début » au démarrage, puis se divise par deux à chaque étape et devient plus petite que 1, donc égale à 0, puisque ce sont des nombres entiers. L'algorithme va donc s'arrêter au bout d'un nombre fini, logarithmique, d'étapes. Terminer en tournant la page si elle est trouvée ou sur « pas trouvé » si l'intervalle de recherche est de longueur nulle sans avoir trouvé la page recherchée.

Posons-nous maintenant une autre question : dans quelle mesure ce mécanisme algorithmique est-il générique ? Regardons un tout autre exemple pour s'en convaincre.

CONVERTIR UNE TENSION ÉLECTRIQUE EN VALEUR NUMÉRIQUE

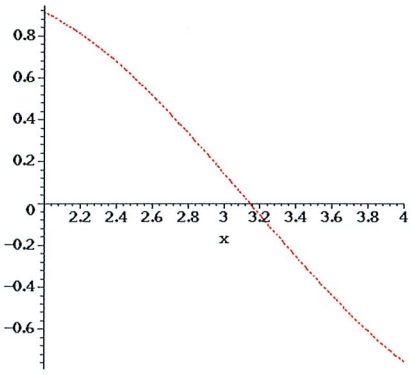
Pour « deviner » la valeur d'une tension électrique continue, un ordinateur numérique doit, en général, comparer cette valeur à une valeur de référence qu'il va produire en sortie pour, de proche en proche, cerner cette valeur, comme le montre le diagramme ci-dessous :



Là encore, la façon de fonctionner de ces convertisseurs à approximations successives est de procéder de manière dichotomique, en divisant l'espace de recherche de deux en deux. Cela permet d'atteindre très rapidement les précisions requises, le millième en dix étapes, le millionième en vingt, comme nous l'avons vu précédemment.

TROUVER LE ZÉRO D'UNE FONCTION MONOTONE DANS UN INTERVALLE

Tous ces problèmes sont reliés au problème mathématique suivant : résoudre une équation de la forme $f(x) = 0$, $\min < x < x \max$ où $f()$ est une fonction continue monotone dans l'intervalle [min, max]. Elle a donc une solution, puisqu'il y a bijection vers un intervalle réel. Si elle change de signe dans cet intervalle $f(\min)\, f(\max) < 0$, elle a une solution unique. C'est le cas de la fonction $\sin(x)$ dans l'intervalle [2, 4], où elle s'annule en π, comme le montre la figure ci-dessous :



Là encore, le même mécanisme de dichotomie permet de résoudre le problème, comme dans l'algorithme suivant :

```
zéro (f) {
    min = début ; max = fin ;
    si f (min) f (max) > = 0, alors renvoyer « pas de solution »
    répéter
        milieu = (min + max) / 2 ;
        si |max - min| < Σ, alors renvoyer milieu
        si f (max) f (milieu) < 0, alors min = milieu
        sinon, si f (min) f (milieu) < 0, alors max = milieu
        sinon renvoyer « pas de solution »
}
```

qui ne diffère du précédent que par la façon de détecter le fait qu'il n'y ait pas de solution et par celle d'effectuer les tests. Il s'agit donc bien d'un principe algorithmique, qui fonctionne à la fois pour des opérations numériques et des opérations symboliques (recherche de mots), comme Ada Lovelace le savait un siècle avant que ne fonctionne le premier ordinateur. Et c'est l'un des mécanismes algorithmiques les plus utilisés.